Faculdade de Engenharia
Universidade do Porto

Performance evaluation of a single core

Parallel and Distributed Computing, Project 1

Class 6
André Moreira up201904721@edu.fe.up.pt
Bruno Mendes up201906166@edu.fe.up.pt
Nuno Alves up201908250@edu.fe.up.pt

Friday 25th March, 2022

Contents

Summary

The central processing unit's cache is rarely a factor in consideration when buying a new machine. However, the cache, with its fast nature and proximity to a processor core, constitutes a main player in the process of accessing memory data: it stores frequently used data, avoiding the need for accessing the slower main memory.

This paper focuses on the impact of accessing contiguous cache memory data, i.e. reading from the same cache line, on the overall performance of data-dependent, heavy algorithms. The different ways of multiplying algebraic matrices are used as a case study.

Algorithms

2.1 Basic Matrix Multiplication

Algorithm 1 Basic Matrix Multiplication

```
1:  function MULT(A, B)
2:      n ← A.rows
3:      C ← new n.n matrix
4:      for i = 1 to n do
5:          for j = 1 to n do
6:              sum ← 0
7:              for k = 1 to n do
8:                  sum ← sum + A_{i,k} * B_{k,j}
9:              end for
10:             C_{i,j} ← sum
11:         end for
12:     end for
13:     return C
14: end function
```

This algorithm is based on calculating the dot product between every line of the A matrix with every column of the B matrix represented as vectors. i is the line of the A matrix, j is the column of the B matrix and k is the kth member of both vectors.

The problem with this algorithm is that the second matrix is traversed vertically. For each access to an element in the B matrix that does not result in a cache hit, some memory located just after $B_{k,j}$ is loaded into the corresponding cache line. After accessing $B_{k,j}$, the next iteration reads from $B_{k+1,j}$, which is not right after $B_{k,j}$. Because this last access is not on the previously loaded cache line, this results in reading from the next cache level or from the main memory, depending on if there is a higher level available. With the increase of the matrices size, these two consecutive accesses will be farther and farther in memory, resulting in a lot of cache misses.

2.2 Line Multiplication

The line multiplication algorithm solves the mentioned problem in the previous algorithm, by switching the order in which we iterate through the k and j values. This will ensure that matrices are always iterated horizontally, resulting in more cache hits.

Algorithm 2 Line Matrix Multiplication

```
1:  function LINE MULT(A, B)
2:      n ← A.rows
3:      C ← new n.n matrix
4:      for i = 1 to n do
5:          for k = 1 to n do
6:              for j = 1 to n do
7:                  C_{i,j} ← C_{i,j} + A_{i,k} * B_{k,j}
8:              end for
9:          end for
10:     end for
11:     return C
12: end function
```

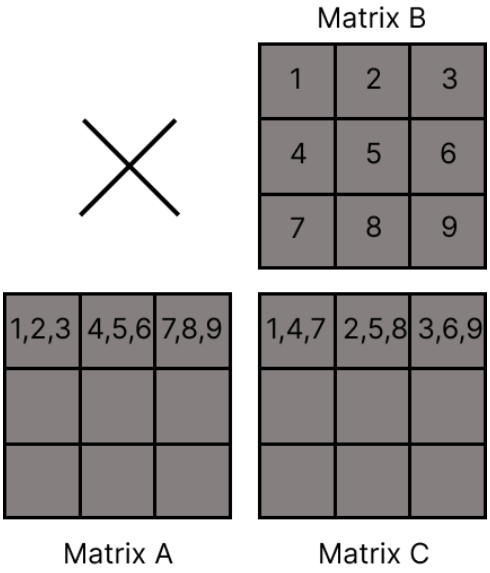

Figure 2.1: Accessed matrix elements on the first 9 iterations of the line multiplication algorithm

As we can see in figure 2.1, where each number identifies the first 9 iterations and memory accesses of the algorithm, all memory accesses are done in adjacent portions of memory, analogous to cache lines, resulting in more cache hits and thus reducing the need for loading new portions of memory into the cache.

2.3 Block Multiplication

Algorithm 3 Block Matrix Multiplication

```
function BLOCK MULT(A, B, b)
    n ← A.rows
    C ← new n.n matrix
    for iBlock = 1 to n, iBlock ← iBlock + b do
        for jBlock = 1 to n, jBlock ← jBlock + b do
            for kBlock = 1 to n, kBlock ← kBlock + b do
                for i = iBlock to iBlock + b do
                    for k = kBlock to kBlock + b do
                        for j = jBlock to jBlock + b do
                            C_{i,j} ← C_{i,j} + A_{i,k} * B_{k,j}
                        end for
                    end for
                end for
            end for
        end for
    end for
    return C
end function
```

The blocked version of the algorithm fixes a problem known as cache thrashing, which happens when a program starts running slow due to filling up the cache. When using the line mult algorithm (2) to calculate the first line of the result matrix the whole B matrix needs to be loaded into memory. If the matrix is too large then the first lines of the B matrix will have already been discarded by the time the second line of the result matrix is being calculated, which results in a cache miss. This (3) blocked version takes advantage of cache lines already accessed by calculating the result in smaller blocks of the matrix.

Performance Results

3.1 Used hardware

The algorithms were tested with mock matrices, with various sizes (and, in the case of block multiplications, for various block sizes, too), on a machine with 8 CPUS Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, each with the following cache layout:

L1d cache: 128 KiB (4 instances)
L1i cache: 128 KiB (4 instances)
L2 cache: 1 MiB (4 instances)
L3 cache: 8 MiB (1 instance)

The produced algorithms do not make use of parallelism, so the number of cores is not a relevant specification.

3.2 Collected metrics

The collected metrics are shown below. One can also run the tests on their machine, selecting option 4 of the C++ program.

Basic Matrix Multiplication

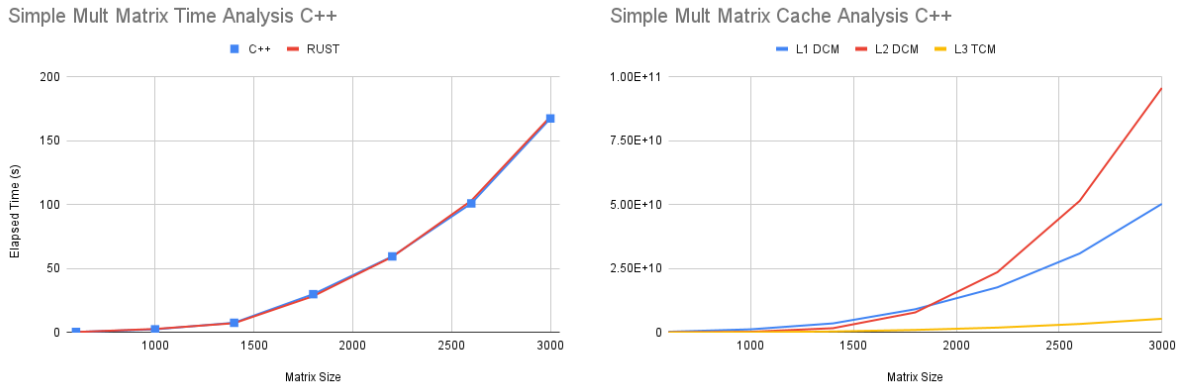

Figure 3.1: Analysis of the simple multiplication algorithm regarding time and cache performance

Line Multiplication

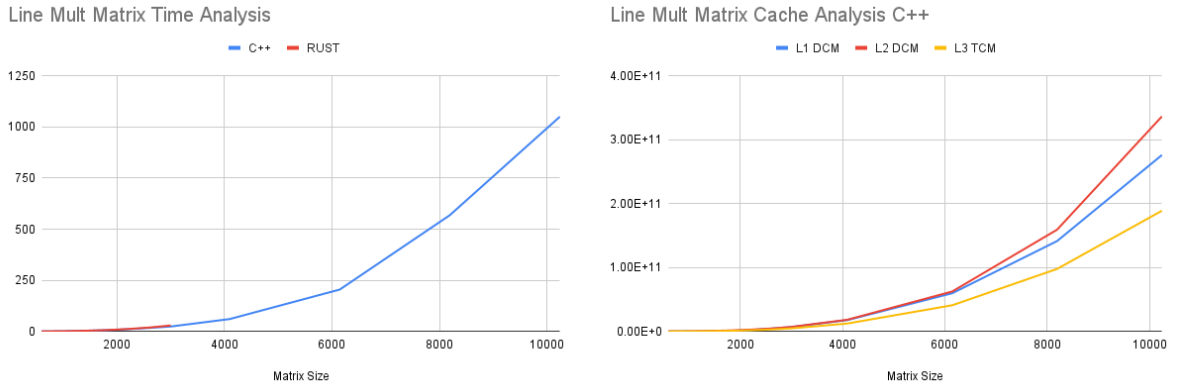

Figure 3.2: Analysis of the line multiplication algorithm regarding time and cache performance

The graphics presented in 3.1 and 3.2 show that the execution times between Rust and C++ are roughly the same. Also, L2 is consistently the cache level with the most number of misses.

Block Multiplication

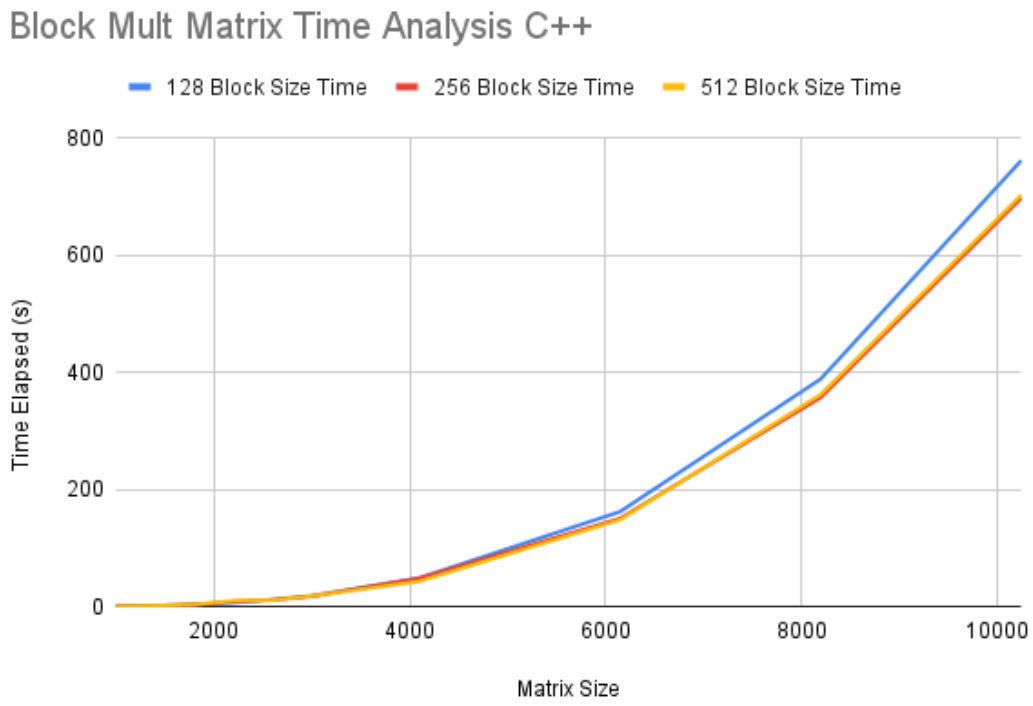

Figure 3.3: Analysis of the block multiplication algorithm regarding time for different block sizes

3.3 Comparison

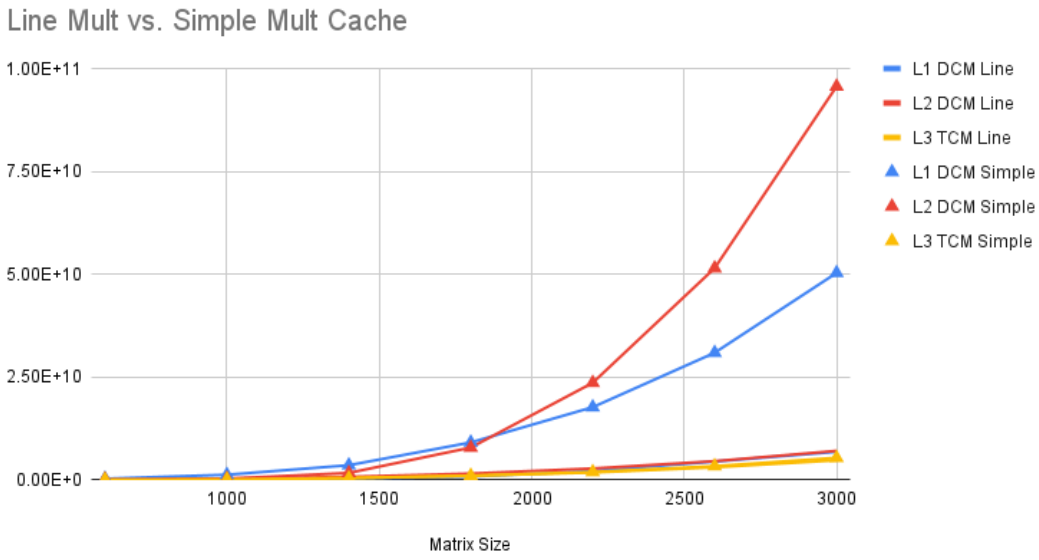

Figure 3.4: Analysis of both line and simple algorithms regarding cache performance

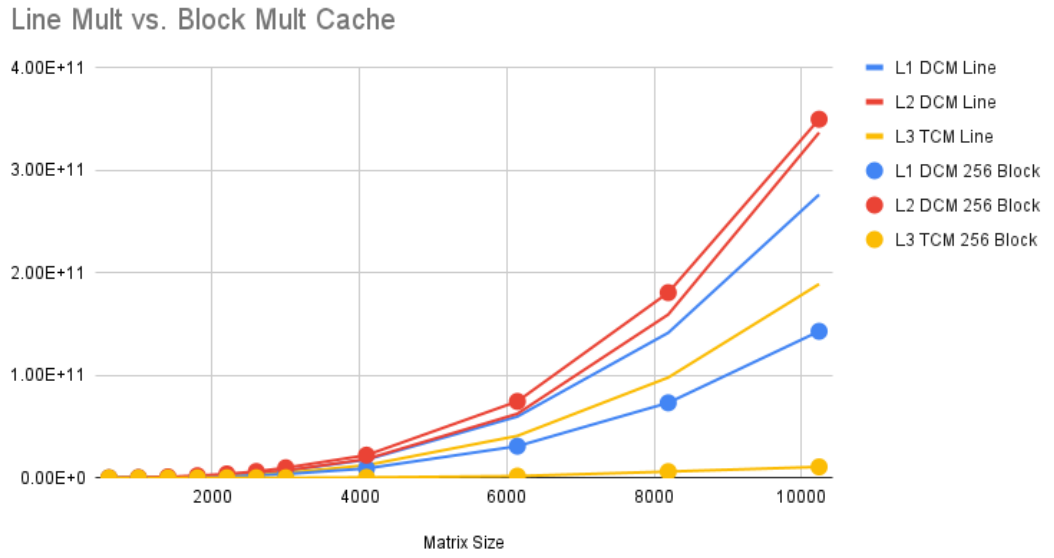

Figure 3.5: Analysis of both block and line algorithms regarding cache performance

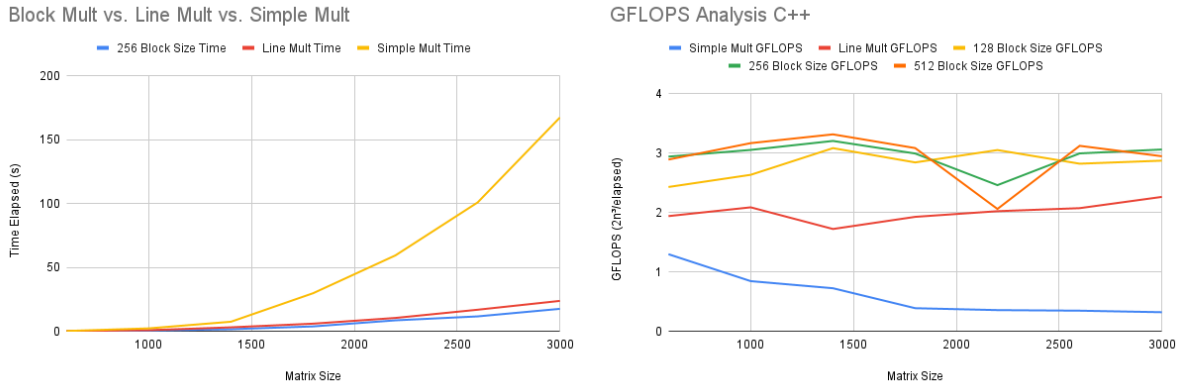

Figure 3.6: Analysis of all algorithms regarding time and floating point operations per second

As expected, the basic matrix multiplication performs considerably worse than the line variant. The block variant comes in handy with very large matrices as it drastically lowered L3 cache accesses because of the reduction of cache thrashing, as we can see in 3.5. With the increase of cache hits, the algorithm becomes more efficient, which is clearly shown in 3.6.

Conclusion

The different algorithms available to compute the product on two algebraic matrices proved to be very distinct in terms of efficiency, depending of how aligned memory access is with the way the CPU loads data in its cache.

It is essential for a programmer to be aware of this phenomenon. Accessing the cache in an efficient way is not something that a compiler is able to enforce, and may constitute a not so obvious and important optimization to heavy programs, unrelated to parallelism.